A CAD Development Strategy for the Next Years

Antonio Rodríguez-Goñi, ETSIN(UPM), Madrid/Spain, antonio.rodriguez@upm.es
Leonardo Fernández-Jambrina, ETSIN(UPM), Madrid/Spain, leonardo.fernandez@upm.es

Abstract

This paper suggests a new strategy to develop CAD applications taking into account some of the most interesting proposals which have recently appeared in the technology development arena. Programming languages, operating systems, user devices, software architecture, user interfaces and user experience are among the elements which are considered for a new development framework. This strategy considers the organizational and architectural aspects of the CAD application together with the development framework. The architectural and organizational aspects are based on the programmed design concept, which can be implemented by means of a three-level software architecture. These levels are the conceptual level based on a declarative language, the mathematical level based on the geometric formulation of the product model and the visual level based on the polyhedral representation of the model as required by the graphic card. The development framework which has been considered is Windows 8. This operating system offers three development environments, one for web applications (HTML5 + CSS3 + JavaScript), and other for native applications (C/C++) and of course yet another for .NET applications (C#, VB, F#, etc.). The user interface and user experience for non-web application is described with XAML (a well known declarative XML language) and the 3D API for games and design applications is DirectX. Additionally, Windows 8 facilitates the use of hybrid solutions, in which native and managed code can interoperate easily. Some of the most remarkable advantages of this strategy are the possibility of targeting both desktop and touch screen devices with the same development framework, the usage of several programming paradigms to apply the most appropriate language to each domain and the multilevel segmentation of developers and designers to facilitate the implementation of an open network of collaborators.

1. Introduction

This paper describes a combination of push-pull technology innovations in the CAD development process to create a new category of applications. Push innovation is the process of incorporating new product concepts to develop truly unique product offerings. On the other hand, pull innovation consist of playing the role of early adopters to integrate innovative solutions which have been developed by third parties.

The push innovation considered in this paper is the programmed design concept, *Rodríguez and Fernández-Jambrina (2012)*. Programmed design is an evolution of parametric design, being its objective to create parametric designs tools. Programmed design provides a methodology to extract the design knowledge from a design experience to store and reuse it many times.

Programmed design must be supported by a comprehensive product model in order to face the modeling of any type of ship. Additionally, the product model has to be created by means of a design language. The main purpose of the language is to publish the modeling algorithms of the CAD application in the designer knowledge domain to let the designer create parametric model scripts.

The pull innovation which is being considered in this paper is the adoption of Windows 8 as a development framework. Windows 8 comes with a new development model to create a new kind of application called Windows 8 Style, which provides a first-class user experience with multi-touch and sensors, being very sensitive in terms of user interface responsiveness. Additionally, Windows 8 still allows developing traditional desktop applications. To cope with these two environments, Windows 8 provides two fundamental stacks of technologies, one for Windows 8 style applications and one for desktop applications, that can be used side-by-side.

Consequently, Windows 8 is the first operating system that tries to get the best of both worlds by providing a mix of the two UI paradigms. The well-known desktop applications are best suited for content-creating scenarios (such as designing CAD product models) whereas Windows 8 style apps are best at consuming content. These type of applications can be very useful when exploiting the aforementioned CAD product models far from the design office, for example, replacing the use of paper drawings by using smart (and sometimes ruggedized) devices. The next paragraphs are devoted to explain these innovations more in detail.

2. Programmed design

2.1. Concept

Programmed design is a proposal to provide advanced users with a CAD environment in which they can incorporate their knowledge in the application by themselves. Consequently, the programmed design functionality is a component to be built on top of an existing CAD application, where the CAD application provides the algorithms used to create the product model elements. To reach this goal, programmed design incorporates a design language to allow the advanced user to write modeling scripts. The CAD application plays the role of provider (of the modeling algorithms) and the design language publishes these algorithms in the designer knowledge domain.

2.2. Design language

A suitable language for supporting modeling tasks performed by a CAD user without programming skills must comply with some requirements. As the language has to be read by the design application, it plays the role of interpreter of the language.

As the CAD user is not supposed to have special skills to write complex algorithms, his objective would be to specify what the program should accomplish, rather than describing how to go about accomplishing it. Additionally, the language should be specifically oriented to the ship design modeling domain. Consequently, the design language should be a declarative interpreted domain specific language with some control sentences such as loops and conditional branches to facilitate executing repetitive tasks and automation of processes.

A declarative language script consists of a sequence of declarations with the objective of building product model elements. Each of these sentences creates a geometric construction which can be either a product model component or an auxiliary entity to be used later to build a more complex component of the product model. These declarative sentences can be considered “constructors” of primitives (auxiliary elements) or components (constitutive product model elements).

Hence, a design language script is a set of constructors which can be combined with control sentences to create loops, branches and procedures. The outcome of executing this script should be a complete ship product model or a part of it.

2.2.1. Types and constructors

The design language has to provide a complete set of element types in order to be able of creating any primitive or component which may be required to build the product model.

According to these language premises, each type contains a set of constructors, each of them implementing a specific algorithm or method to create an element of such type. Each constructor must be identified by a unique name and has associated a list of arguments which collects all the input data required by the algorithm. Each element of the list of arguments is a primitive of the product model. The result of executing the constructor is a new primitive or a new component of the product model.

In order to create a primitive or a component of a concrete type with a specific constructor, all of the primitives required by the list of arguments must be created previously. This can lead to very heavy scripts. To avoid this problem the language must provide the option of specifying anonymous constructors. An anonymous constructor creates on the fly an element of the required type within the list of arguments of another constructor, in order to be consumed immediately and for this reason no name is required.

Then the syntax of a constructor invocation is: *TYPE elemID ConstructorID (primID1, . . . , primIDn);*

An anonymous constructor contains only the constructor identification with the list of arguments, as the type is inferred from the context:

ConstructorID (primID1, . . . , primIDn)

Using an anonymous constructor consists in substituting a primitive identification in the list of arguments of another constructor with the anonymous constructor invocation:

TYPE elemID ConstructorID1(primID1,primID2, . . . , primIDn);

↓

TYPE elemID ConstructorID1(primID1, ConstructorID2(p1, . . . , pn), . . . , primIDn);

Anonymous constructors can be nested ad infinitum.

3.3.2. Control sentences

The designer must have the possibility of writing loops and conditional jumps to implement more advanced procedures. The following schemas are required:

Conditional branching: *if (B1);. . . ;else if (B2);. . . ;else;. . .;end if;*

Conditional loop: *while(B);. . .;end while;*

In the above expressions, B, B1 and B2 are identifications of Boolean elements or anonymous constructors of such type of primitive.

List scanning loop: *for(listID, listIndex, listElement);. . . ; end for;*

In order to take full advantage of the list scanning loop, the language has to include a list type for each type of primitive (float list, points list, curves list, etc.) and some list of lists types (list of lists of floats, etc.).

Element mutator: *set elemID ConstructorID (primID1,. . . , primIDn);*

The mutator syntax is required for modifying existing elements by means of any of its constructors.

In order to organize, encapsulate and reuse design language scripts, the language must provide the possibility of writing procedures and user constructors:

Procedure encapsulation: *proc procID(inputID1,.., inputIDn);. . .;end proc(outputID1,.., outputIDm);*

Procedure invocation: *call proc((inputID1,.., inputIDn), (outputID1,.., outputIDm));*

User constructor definition: *cons TYPE ConstructorID(TYPE1 primID1,.., TYPEn primIDn);. . . ;ret retID;*

User constructor invocation: *TYPE elemID ConstructorID(primID1, . . . , primIDn);*

A complete language specification is out of scope of this paper, but in order to explore the programmed design concept the authors have developed a program for demonstration purposes. This prototype implements some geometric constructors for hull form design. In the next paragraph a simplified case of use of programmed design is shown, which has been created with the demonstration program just to illustrate this paper.

2.3. Example

The first example illustrates a script for the creation of a fore profile curve from the given x and z coordinates.

lf ***xp*** .(50,59,55,65,70);	Create a list of floats (lf type) named ***xp*** with the default constructor
lf ***zp*** .(0,4,10,17.5,20);	Create a list of floats (lf type) named ***zp*** with the default constructor
lp2 ***lprof*** .lf (***xp***, ***zp***);	Create a list of 2D points (lp2 type) named ***lprof*** with the .lf constructor from two list of floats (one for the u coordinates and the other for the v coordinates)
c2 ***c2prof*** .(***lprof***);	Create a 2D curve named ***c2prof*** with the default constructor from a list of 2D points
c ***prof*** .xz(0, ***c2prof***);	Create a 3D curve named ***prof*** with the .xz constructor from y coordinate and a 2D curve (if y is omitted default value is 0)

The following picture illustrates the first example with some syntactic simplifications easily admitted by the language, including the use of anonymous constructors.

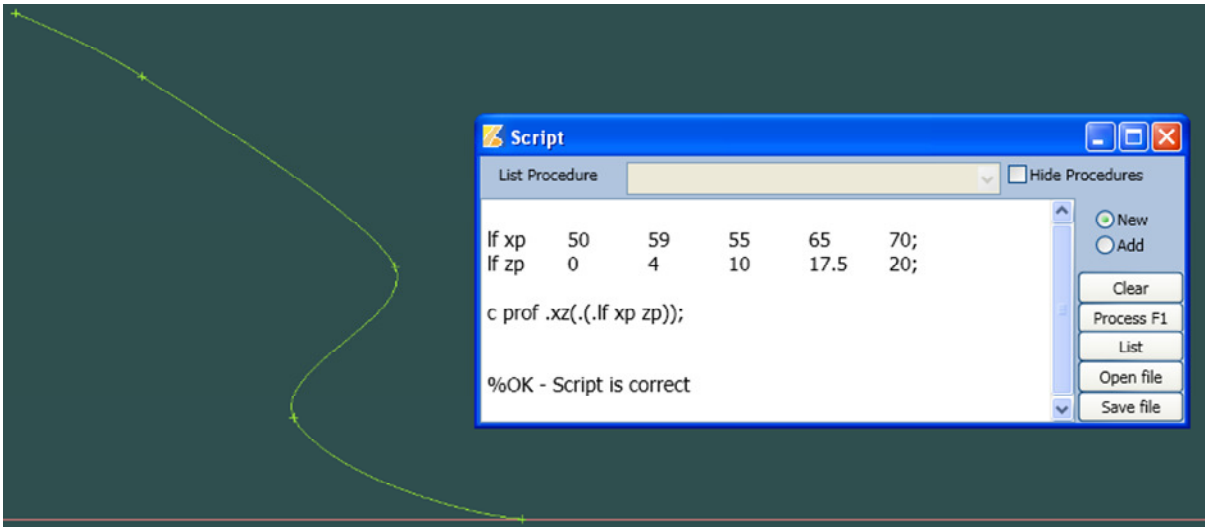

Fig.1: Fore profile curve definition (simplified)

The construction of the curve can be enhanced by considering the tangencies at each point of the curve. Another constructor for the 2D curve is required for this new definition. The following script shows only the modifications from the previous one.

ltc2 ***tprof*** .(0,90,90,30, .n);	Create a 2D curve tangencies list named ***tprof*** with the default constructor from a list of 2D tangencies (angles). A free tangency is indicated with the tangency constructor .n
c2 ***c2prof*** .pt(***lprof, tprof***);	Create a 2D curve named ***c2prof*** with the .pt constructor from a list of 2D points and a list of 2D tangencies
c ***prof*** .xz(0, ***c2prof***);	Create a 3D curve named ***prof*** with the .xz constructor from y coordinate and a 2D curve (if y is omitted default value is 0)

Fig. 2 illustrates the new version of the curve with the same syntactic simplifications.

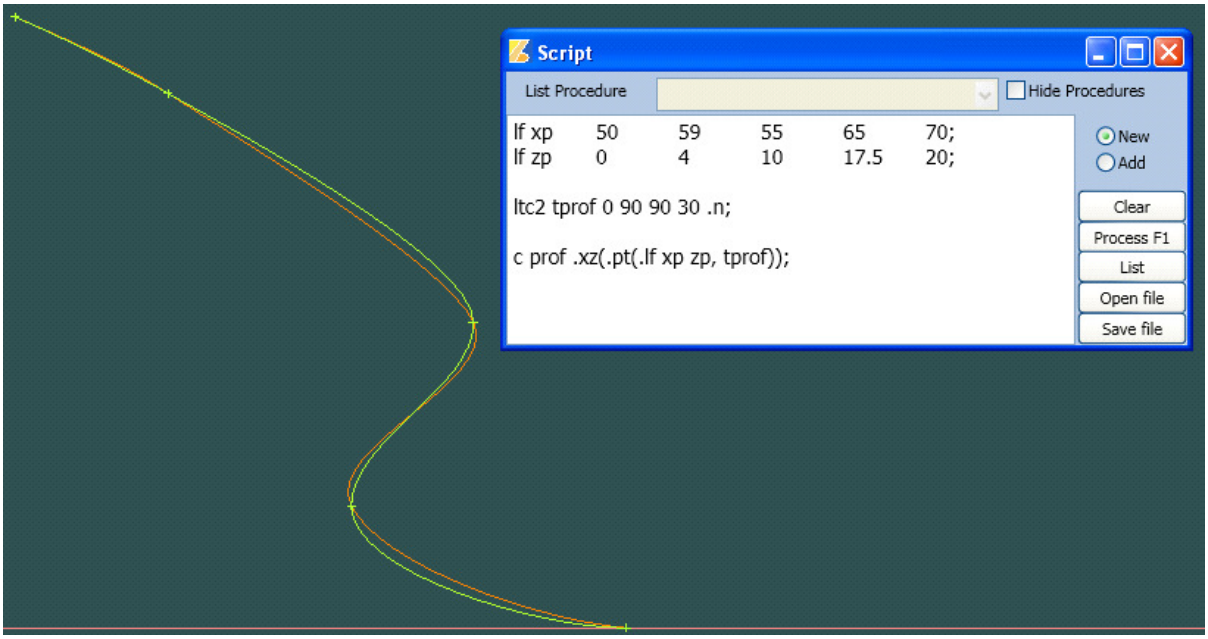

Fig.2: Fore profile curve definition (enhanced)

To illustrate the use of loops, the next example shows the construction of several frames from a simplified offset. The offset is defined by means of list of (lists of) floats.

lf ***x*** .(0, 20, 40);	List of floats (lf) for frame abscissas named ***x***
lf ***z*** .(0,4,10,17.5,20);	List of floats (lf) for waterline heights named ***z***
mf ***y*** .(.(0,10,17.5,20,20), .(0,7.5,16,19.5,19.5), .(0,4.5,9.5,15,15));	List of lists of floats for frame half breadths, named ***y***. There is one list of half breadths for each frame (3 lists = length of ***x*** list), with the half breadths for each height (5 half breadths = length of ***z*** list).
ltc2 ***tframes*** .(0, .n, .n, 90, 90);	List of 2D curve tangencies named ***tframes***, for the frames
for ***x i xc***;	Init loop through list ***x*** using index ***i*** =(0, 1, 2) and the list value at ***i*** named ***xc*** =(0, 20, 40)
c **fr<i>** .yz(***xc***, .pt(.lf(.lis ***y i, z***), ***tframes***));	Create a 3D curve named ***fr<i>*** =(***fr0, fr1, fr2***) with the .yz constructor from x coordinate and a 2D curve
end for;	End loop

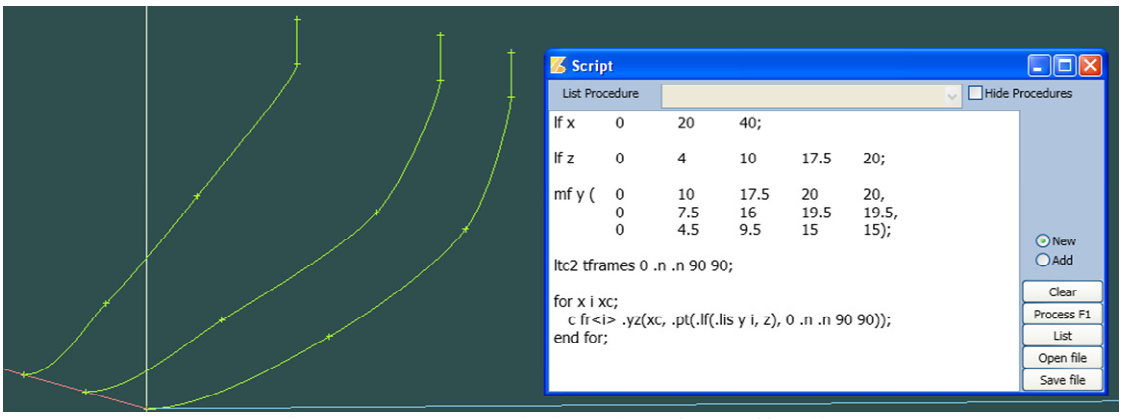

Fig.3: Frames created with a loop from offset lists

Before creating a patch with the previous curves, a patch tangency is required for the fore end. The creation of this curve from the fore profile is illustrated in the following example.

ltc2 ***tprof*** .(0,90,90,30, .n);	Create a list of 2D tangencies named ***tprof*** with the default constructor using a list of angles. A free tangency is indicated with the tangency constructor .n
c2 ***c2prof*** .pt(***lprof, tprof***);	Create a 2D curve named ***c2prof*** with the .pt constructor from a list of 2D points and a list of 2D tangencies
c ***prof*** .xz(0, ***c2prof***);	Create a 3D curve named ***prof*** with the .xz constructor from y coordinate and a 2D curve (if y is omitted default value is 0)
pla ***pi*** .xy(50 0, 20);	Create a plane named ***pi*** which intersection with XY plane passes through XY point (50, 0) forming 20º with X axis
c ***tfore*** .pxz(***pi***, ***c2prof***);	Create a 3D curve contained in the plane ***pi*** and which projection in the XZ plane is ***c2prof***

Both curves, the fore profile and the fore end tangency, are created with the same geometry in order to make waterline endings perpendicular to the centre plane. Fig. 4 illustrates the example with the simplified notation.

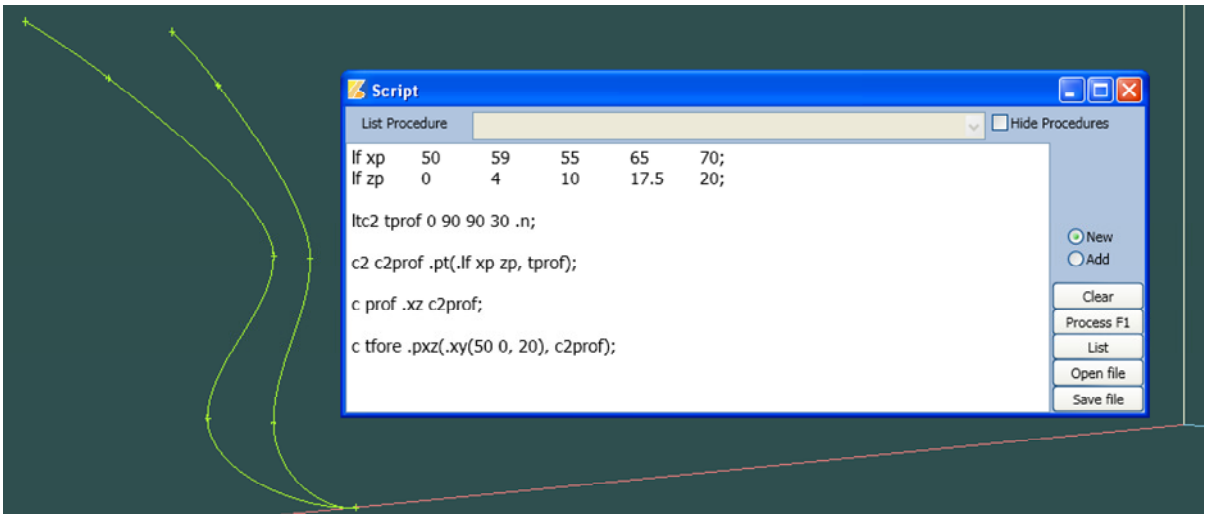

Fig.4: Fore end tangency curve

Then, the fore body can be created using a list of curves in order to gather all of the input data required by the patch constructor.

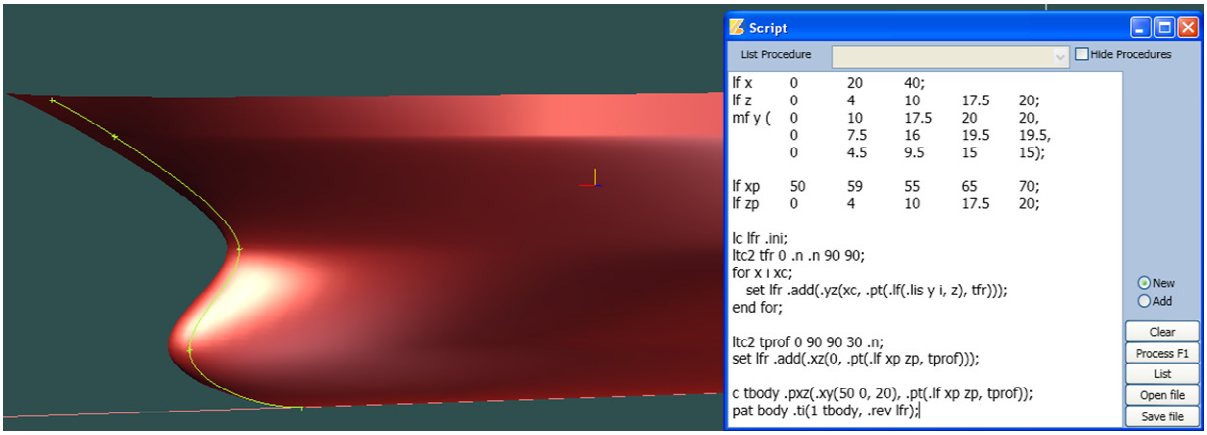

Fig.5: Fore body creation

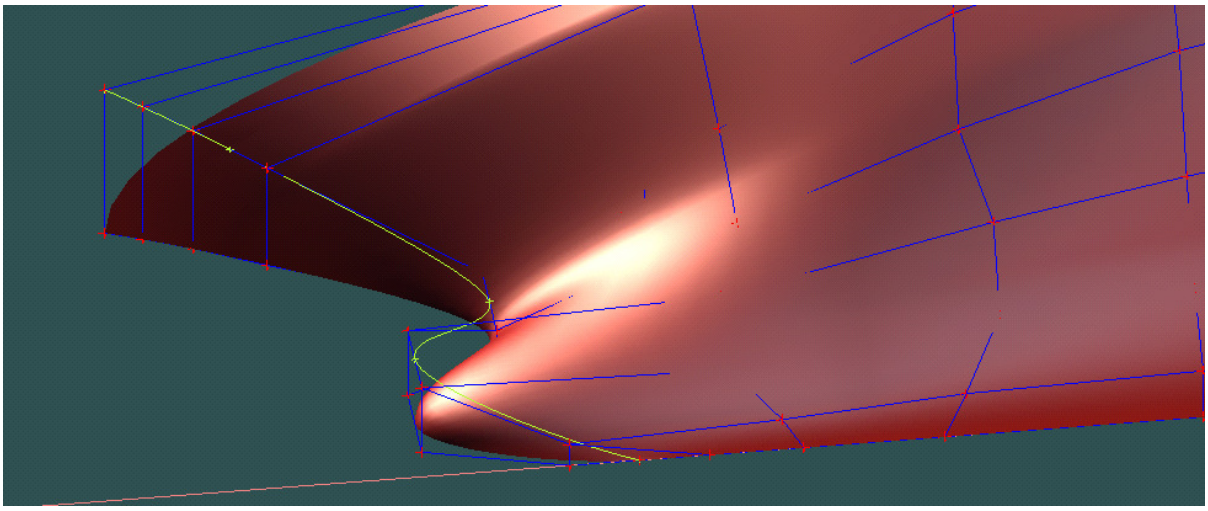

Fig.6: Fore end construction with the tangency curve

The script shown in Fig.7 generates a product model (or a part of it) with the geometric representation of the ship hull forms. At this point the script can be considered as a conceptual representation of the product model as it contains all the information required to generate it. One of the major advantages of programmed design is that it comes with the geometric parameterization out of the box. Any of the values used within the script to generate the product model can be converted automatically in a parameter to produce variations of the model. To illustrate this feature, in the following example the complete script is encapsulated within a user constructor which exposes only one parameter to produce model variations. For this example, the bulb nose abscissa has been selected as a parameter.

```
const pat .bulb(f len);

   lf x 0 20 40;
   lf z 0 4 10 17.5 20;
   mf y ( 0 10 17.5 20 20, 0 7.5 16 19.5 19.5, 0 4.5 9.5 15 15);

   lf xp 50 len 55 65 70;
   lf zp 0 4 10 17.5 20;

   lc lfr .ini;
   ltc2 tfr 0 .n .n 90 90;

   for x i xc;
      set lfr .add(.yz(xc, .pt(.lf(.lis y i, z), tfr)));
   end for;

   ltc2 tprof 0 90 90 30 .n;
   set lfr .add(.xz(0, .pt(.lf xp zp, tprof)));
   c tbody .pxz(.xy(50 0, 20), .pt(.lf xp zp, tprof));

   pat body .ti(1 tbody, .rev lfr);

ret  body;
```

Fig.7: User constructor defining the patch with all the input data as internal values and exposing the abscissa of the nose of the bulb as a single parameter or external value

2.4. Programmed design architecture

The different types of algorithms and services which are implemented by the product model can be organized into three levels of abstraction. The most abstract level is devoted to implement the design language, the organization of the model, the topology and any other kind of relational or organizational aspect of the product model. The following level provides a mathematical formulation for each of the entities created by the product model. Finally, the model has to interact with the graphic card, which requires a visual representation of the model based on faceted or polyhedral surfaces and polylines. This visual model is created from the mathematical formulation by means of some algorithms which require the selection of certain precision parameters.

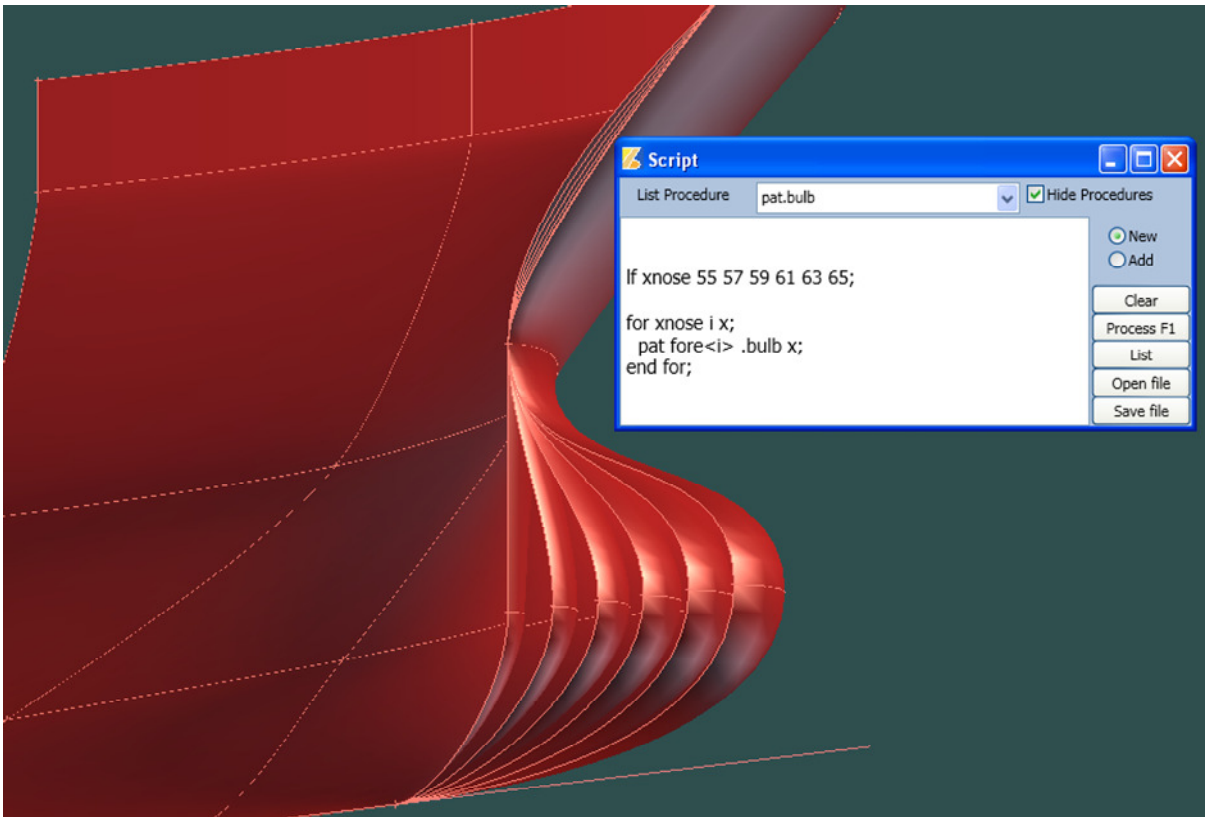

Fig.8: Product model variations produced with a list of abscissas and a loop over it to generate one patch for each abscissa of the bulb nose

These three levels can be identified as the conceptual level, the mathematical level and the visual level and they can be interpreted as a cause–effect chain or a three stage projection process. The concept is implemented or described mathematically and these mathematical entities are visualized or represented in the machine. Ideally, the whole definition of the product model should be performed at the conceptual level while the whole product model exploitation would take place at the visual level.

2.5. Use cases

Programmed design can be considered as a tool for creating design tools or as a method to store and reuse design experiences. For a single or sporadic design, programmed design may not be the preferred tool because the user interface based on a design language is not the most adequate for the occasional designer. To fill this gap, the scripts developed with programmed design could be wrapped within advanced user interface widgets to facilitate their usage by less experienced designers.

Another scenario where programmed design provides some advantages is CAD-PLM integration. With programmed design CAD apps, the PLM can manage the product model by controlling scripting files instead of product model files. CAD-PLM integration can be easily implemented at the conceptual level.

Finally, another way to improve the use of programmed design is to reverse the normal flow from conceptual level to mathematical level. Hence, this functionality would provide the possibility of transferring a mathematical model to a conceptual model. This process could require the selection of the types of entities and constructors which are required to generate the mathematical model being imported, but it should be automated as much as possible in order to take full advantage of the process. This improvement would allow the application to incorporate external designs with the same capabilities as native or proprietary designs and consequently create programmed designs with external information in a very easy way.

3. Development Framework

3.1 Windows 8

Windows 8 comes with two fundamental stacks of technologies that can be used side-by-side, one for Windows 8 style applications, which is adequate for touch screen devices and other for the "traditional" desktop applications, *Novák et al. (2012)*. While Windows 8 has put a lot of importance on the user interacting with the computer through the use of touch gestures, the Windows 8 user interface is referred to as touch-first and not as touch-only. Fig. 9 shows both stacks of technologies, showing their different layers. Web application layers have been omitted to simplify the figure.

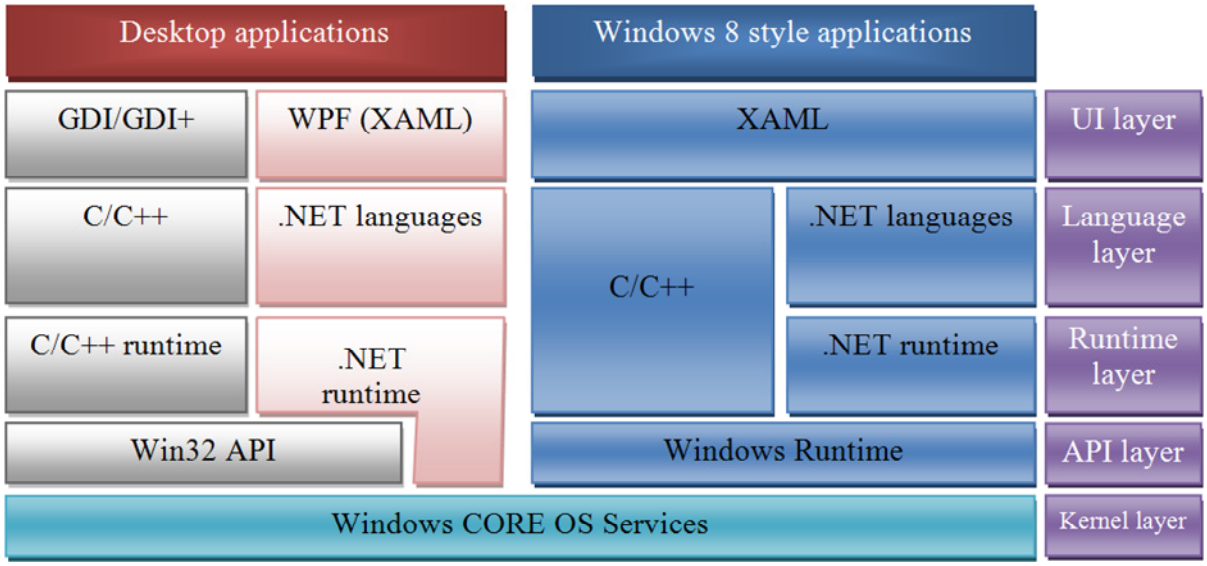

Fig. 9: Windows 8 stacks of technologies for desktop and Windows 8 style applications, adapted to CAD applications development

The advantage of this new framework is that both architectures can share the same business layers (the business layer can be located within the language layer of the Fig.8) while their differences can be isolated in the UI and API layers. The UI layer of both types of application can be based on the same declarative UI language, XAML. Microsoft provides several implementations of XAML. WPF is the XAML implementation for desktop applications while there is a new implementation for Windows 8 style applications called Windows 8 XAML. Other implementations are the different versions of Silverlight.

The API layer for native desktop applications is the traditional Win32 API, while managed desktop applications use the .NET framework API which is built on top of the previous one. Windows 8 style applications have one single API layer, Windows Runtime, which can be accessed from C++, C#, Visual Basic, and JavaScript.

3.2. Language layer

For the past couple of decades, object-oriented programming has dominated the CAD application development industry due to its ability to hide complexity and provide structure and intuition. Object-oriented programming encourages code reuse and provides specialization through inheritance, which can help to deal with complexity. The most widespread development language in this context is C++. This language is extremely popular and therefore lots of support is available. But C++ has a more difficult learning curve than modern object oriented languages like java and C#. The C++ language is very demanding about how code is formatted and the most powerful features, such as templates, have a very complex syntax. In C++ GUI, threads and networking are not standardized, requiring the use of non-standard third-party libraries. Memory management in C++ is quite a complex feature compared

to java and C#, which frees developers from manually allocation and freeing the memory occupied by objects.

Taking into account all these facts, C# can be considered a serious alternative to C++. In the comparison between C# and C++, it is usually accepted that C++ provides better program performance while C# provides better programmer productivity. The CAD applications require intensive use of graphic operations, where the C++ performance can be decisive. To counteract this advantage, the C# JIT (just in time) compiler has incorporated lots of optimizations and really the differences in performance for a CAD application could be relevant only the first time the program is executed and this can be assumed as the last deployment step. Just focusing on the development process issues (productivity, code readability, code free of errors, memory leaks, ease of maintenance, etc.) C# has many advantages.

One point in favor of C++ is that the cloud is promoting native applications because they reduce battery consumption with respect to managed applications. This is very important when applications are executed in smart devices. Maybe this is the reason why Windows 8 gives to C++ such a relevant role in its development architecture. But using C++ or C# should not be an issue in Windows 8, because in this platform both languages can coexist without too much effort if a clear frontier between them is well established. In desktop applications this interface is built with technologies such as platform invocation (P/Invoke) and COM interoperability. For Windows 8 style applications there is a smarter solution in which native and managed code can interoperate easily because each programming language is capable of consuming Windows Runtime objects. In order to utilize a component in any other language, its functionality can be exposed creating a reusable Windows Runtime object that can be utilized in Windows 8 applications independently of the consuming programming language. An application built on this basis is called a “hybrid solution”.

C++ and C# are both object oriented languages. While object-oriented programming may work well for modeling some concepts, it has a tough time encoding algorithms and abstract concepts because not all kinds of complexity submit willingly to the mechanisms of encapsulated shared state and virtual methods. In the context of object-oriented programming, the solution comes in the form of general reusable solutions called design patterns, which formalize best practices and help to describe abstractions in terms of objects and their relationships, *Smith (2012).*

While design patterns are helpful, they are simply a compensation for object-oriented programming inability to simply express certain concepts. In addition, they are a burden for the programmer by forcing the creation of boilerplate code in order to define the contract and relationships between objects. When reaching this complexity, the alternative to object-oriented programming is functional programming. Functional programming is a paradigm originating from ideas older than the first computers. Its history goes as far back as the 1930s, when Alonzo Church and Stephen C. Kleene introduced a theory called lambda calculus as part of their investigation of the foundations of mathematics. Even though it did not fulfill their original expectations, the theory is still used in some branches of logic and has evolved into a useful theory of computation, *Petricek and Skeert (2012).*

Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values. Functional languages are expressive, accomplishing great feats using short, succinct, and readable code. All of this is possible because functional languages provide richer ways for expressing abstractions. Developers can hide how the code executes and specify only the desired results. The code that specifies how to achieve the results is written only once.

As a result, many mainstream languages now include some functional features. In the .NET world, generics in C# 2.0 were heavily influenced by functional languages. One of the most fundamental features of functional languages is the ability to create function values on the fly, without declaring them in advance. This is exactly what C# 2.0 enables to do using anonymous methods, and C# 3.0

makes it even easier with lambda expressions. The whole LINQ project is rooted in functional programming. Also C++ 11 brings a number of new tools for functional-style programming. While the mainstream languages are playing catch-up, truly functional languages have been receiving more attention too. The most significant example of this is F#. F# is a functional programming language for the .NET Framework. It combines the succinct, expressive, and compositional style of functional programming with the runtime, libraries, interoperability, and object model of .NET.

Programming paradigms are not exclusive. The C# language is primarily object-oriented, but in the 3.0 version it supports several functional features. On the other side, F# is primarily a functional language, but it fully supports the .NET object model. The great thing about combining paradigms is that developers can choose the approach that best suits their problem. Functional programs on .NET still use object-oriented design as a methodology for structuring applications and components.

Thanks to functional programming, many of the standard object-oriented design patterns are easy to use because some of them correspond to language features in F# or C# 3.0. Also, some of the design patterns are not needed when the code is implemented under the functional paradigm. The biggest impact of functional programming is at the level where the algorithms and behavior of the application are encoded. Thanks to the combination of a declarative style, succinct syntax, and type inference, functional languages help to express concisely algorithms in a readable way. Another important feature of functional programming is the immutability of data structures (values instead of variables) which leads to concurrency friendly application design. Using a declarative programming style, parallelism can be easily introduced into existing code.

In the context of CAD applications development, functional programming offers significant productivity gains in important application areas such as the implementation of complex algorithms. Additionally, F# supports yet another programming paradigm, the language-oriented programming, *Syme et al. (2012)*. Language-oriented programming facilitates the manipulation and representation of languages using a variety of concrete and abstract representations and it is based on three advanced features of F# programming: F# computation expressions (also called workflows), F# reflection, and F# quotations. This paradigm is very useful to implement the programmed design concept.

3.3. UI & API layers

When the Windows Presentation Foundation (WPF) graphical subsystem was introduced in the .NET Framework the UI development paradigm was totally changed, from imperative programming to declarative programming. In order to describe UI elements, WPF uses the eXtensible Application Markup Language (XAML), an XML language. WPF also leverages the hardware capabilities of graphics processing units (GPUs). Additionally, Silverlight (the Microsoft's rich Internet application framework) also uses XAML to define the user interface, *Novák et al. (2012)*.

In Windows 8, the core of the XAML-based WPF and Silverlight technologies has become a part of the operating system, rewritten in native code. The UI of C++, C#, and Visual Basic applications can be defined in XAML. The same XAML produces the exact same UI in every language, without constraints or barriers. Because of the uniform UI technology and the same APIs providing access to the operating system services, application models are the same for C++, C#, and Visual Basic. XAML not only defines the UI, but can also declare its dynamic behavior using a minimal amount of code. XAML connect the UI with the business layer of an application. Following are some important features of XAML, *Novák et al. (2012)*:

- XAML is designed and tailored to allow creating rich, powerful desktop or Internet applications and to produce a superior user experience. In addition to providing simple UI elements (such as text boxes, buttons, lists, combo boxes, images, and so on), it also provides the freedom to create content with animation and media elements, such as video and audio. In contrast to the traditional UI approach with rectangular UI elements, XAML enables to change the entire face of an application.

- XAML provides a very flexible layout system that makes it easy to create layouts which can be automatically adapted to a number of factors, such as available screen size, number of items displayed, size of the displayed elements, and magnification.

- Styles and templates are features that contribute to a smooth cooperation between developers and designers. Developers implement the logic of the application, so that they never set the visual properties of the UI directly. Instead, they signal programmatically that the state of the UI is changed. Designers create the visuals of the UI, taking into account the possible states of the UI.

- With XAML data-binding, information coming from the database and the application's logic can be declaratively bound to the UI elements. Data binding works in cooperation with styles, templates, layouts, and even animations.

- XAML UI incorporates vector graphics. Vector graphics use math to render the contents, rather than pixels. The components are composed of curves and lines, and not of dots. This means that, with different resolutions, the UI remains visually pleasing. Scaling up and scaling down can be performed easily without the loss of quality. XAML-based UIs always show crisp and perfect fonts and drawings.

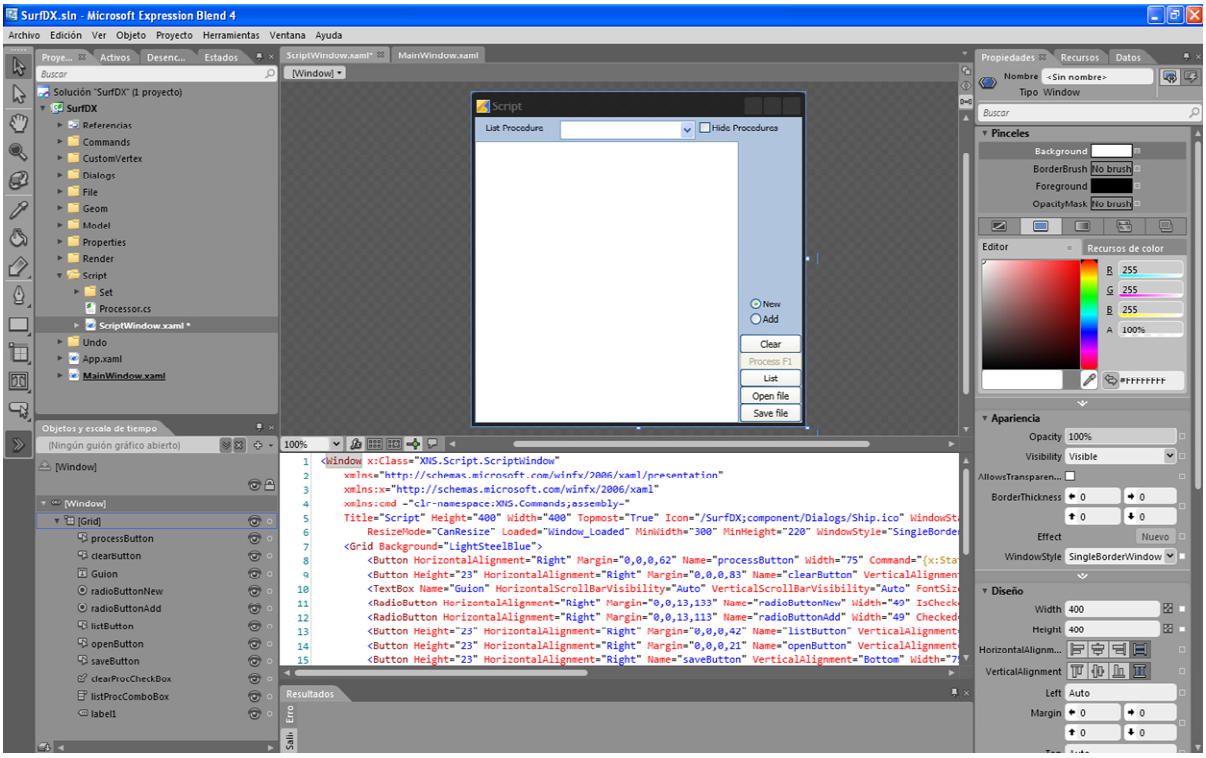

Fig. 10: Expression Blend (XAML visual edition)

XAML approach promotes true separation of concerns. Since XAML is its own file type, it is by necessity separated from the code that executes. While there is still a code behind file that accompanies it, good XAML together with solid application architecture can almost eliminate the need for code behind. In order to encourage better architecture and less code behind, XAML provides support for application features such as data binding and commands.

Expression Blend is the graphic design tool of choice for XAML based applications, *Cochran (2012)*. Blend is a vector graphic design tool which translates the design on the screen to the expressive text of XAML, bringing true separation of concerns to visual application development. It is mainly a design tool which can be used by designers and developers to share the product creation process. In

summary, Blend is a visual design tool that generates XAML, increases drastically the productivity, provides rapid prototyping, and greatly simplifies visual tasks.

With respect to the 3D API, it must be noted that everything in Windows 8 is optimized for and built around DirectX, from the developer platforms to the OS and to hardware design. This way, DirectX must be used to achieve the highest performance rendering in Windows 8, but DirectX APIs are available only in C++ and they are not defined as Windows Runtime types. In order to use DirectX from C#, the DirectX calls must be wrapped. For Windows 8 style applications, these calls can be wrapped in a separate Windows Runtime type library written in C++. Then, these Windows Runtime objects can be included and consumed from C#. For desktop applications the wrapper must be based on the interoperability tools already mentioned.

Fortunately, there is a great initiative in the form of an open source project called SharpDX, which has been created by Alexandre Mutel. The following paragraph is extracted from SharpDX web site, *Mutel (2010)*:

"SharpDX is an open-source project delivering the full DirectX API under the .Net platform, allowing the development of high performance game, 2D and 3D graphics rendering as well as real-time sound application. SharpDX is built with a custom tool called SharpGen able to generate automatically a .Net API directly from the DirectX SDK headers, thus allowing a tight mapping with the native API while taking advantages of the great strength of the .Net ecosystem. SharpDX is providing the latest DirectX and Multimedia API to .Net with true AnyCpu assemblies, running on .Net and Mono. SharpDX provides also a high level Graphics API called the Toolkit. SharpDX is ready for next generation DirectX 11.1 API, Windows 8 and Windows Phone 8 Platform. SharpDX is the fastest managed DirectX implementation."

4. Conclusions

This paper is a proposal containing several technical innovations to develop a new generation of CAD applications. The main objective is to incorporate some of the following competitive advantages in the new product:

- Desktop and touch screen applications developed with the same framework and sharing most of the software layers will allow to extend the CAD functionality to smart devices.

- The possibility to use several programming paradigms allows a better segmentation of developers, advanced users and UI designers, providing a very productive separation of concerns which facilitates the implementation of an open network of collaborators:

    - Object oriented programming for developers coding service layers
    - Functional programming for engineers and scientist writing complex algorithms, languages and semantics
    - XAML visual design for UI designers
    - Programmed design for advanced users creating model scripts to store and reuse knowledge and to facilitate PLM integration

References

COCHRAN, J. (2012), *Expression Blend in Action*, Manning Publications

MUTEL, A. (2010), *SharpDX,* http://sharpdx.org/

NOVÁK, I.; ARVAI, Z.; BALÁSSY, G.; FULOP D. (2012), *Beginning Windows 8 Application Development,* Wiley & Sons

PETRICEK, T.; SKEERT, J. (2010), *Real-World Functional Programming*, Manning Publications

REIDAR, T.; RODRIGUEZ, A. (2004), *Automation tools in the design process,* 3rd Int. Conf. Computer and IT Application in the Maritime Industries (COMPIT), Siguenza

RODRIGUEZ, A.; FERNANDEZ-JAMBRINA, L. (2012), *Programmed design of ship forms,* Computer-Aided Design 44, pp.687-696

RODRIGUEZ, A.; GONZALEZ, C.; GURREA, I.; SOLANO, L. (2003), *Kernel architecture for the development of cad/cam applications in shipbuilding environments.* 2nd Int. Conf. Computer and IT Application in the Maritime Industries (COMPIT), Hamburg

RODRIGUEZ, A.; VIVO, M.; VINACUA, A. (2000), *New tools for hull surface modeling* 1st Int. Conf. Computer and IT Application in the Maritime Industries (COMPIT), Potsdam

SMITH, C. (2012), *Programming F#*, O'Reilly Media

SYME, D.; GRANICZ, A.; CISTERTINO, A. (2012), *Expert F# 3.0*, A Press